Chapter 2

Variant's assessment

RQ2. To what extent the generated variants are different ?

In this chapter, we investigate to what extent the artifically created variants are different. We propose a methodology to compare the program variants both statically and during runtime. Besides, we present a novel study on code preservation, demonstrating that the code transformations introduced by CROW are resilient to later compiling transformations during machine code generation. We evaluate the variant's preservation in both existing scenarios for WebAssembly, browsers and standalone engines.

2.1 Metrics

In this section we propose the metrics used along this chapter to answer RQ2. We define the metrics to compare an original program and its variants statically and during runtime. Besides, we proposed the metrics to compare program variants preservation.

2.1.1 Static

To measure the static difference between programs, we compare their bytecode instructions using a global alignment approach. In a previous work of us we empirically demonstrated that programs semantic can be detected out of its natural diversity [?] . We compare the WebAssembly of each program and its variant using Dynamic Time Warping (DTW) [**?**]. DTW computes the global alignment between two sequences. It returns a value capturing the cost of this alignment, which is actually a distance metric. The larger the DTW distance, the more different the two sequences are.

TODO Add and example here ?

Metric 1 *dt_static: Given two programs P_X and V_X written in X code, dt_static(P_X, V_X), computes the DTW distance between the corresponding program instructions for representation X.*

A dt_static(P_X, V_X) of 0 means that the code of both the original program and the variant is the same, i.e., they are statically identical in the representation X. The higher the value of dt_static, the more different the programs are in representation X.

Notice that for comparing WebAssembly programs and its variants, the metric is the instantiation of dt_static with $X = WebAssembly$.

2.1.2 Program traces and execution times

We measure the difference between programs at runtime by evaluating their execution trace, at function and instruction level. Also, we include the measuring of the execution time of the programs. Besides, we compare their execution times.
TODO Replace and explain the stack trace as stack operations

Metric 2 *dt_dyn: Given a program P, a CROW generated variant P' and T a trace space ($T \in \{Function, Instruction\}$) dt_dyn(P,P',T), computes the DTW distance between the traces collected during their execution in the T space. A dt_dyn of 0 means that both traces are identical.*

The higher the value, the more different the traces.

Metric 3 *Execution time: Given a WebAssembly program P, the execution time is the time spent to execute the binary.*

2.1.3 Variants preservation

The last metric is needed because WebAssembly is an intermediate language and compilers use it to produce machine code. For program variants, this means that compilers can undo artificial introduced transformations, for example, through optimization passes. When a code transformation is maintained from the first time it is introduced to the final machine code generation is a preserved variant.

Part of the contributions of this thesis are our strategies to prevent reversion of code transformations. We take engineering decision regarding this in all the stages of the CROW workflow. We disable all optimizations inside CROW in the generation of the WebAssembly binaries. This prevents the LLVM toolchain used to remove some introduced transformations. However, the LLVM toolchain applies optimizations by default, such as constant folding or logical operations' normalization. As we illustrate previously, these are some transformations found and applied by CROW. We modified the LLVM backend for WebAssembly to avoid this reversion during the creation of Wasm binaries. This phenomenon is sometimes bypassed

by diversification studies when they are conducted at high-level. As another contribution, we conduct a study on preservation for both scenarios where Wasm is used, browsers and standalone engines. In

The final metric corresponds to the preservation study. We compare two programs to be different under the WebAssembly representation and under the machine code representation after they are compiled through a collection of selected WebAssembly engines. We use two instances of Metric 1 for two different code representations, WebAssembly and x86. The key property we consider is as follows:

Property 1 *Preservation: Given a program P and a CROW generated variant P', if dt_static(P_{Wasm}, P'_{Wasm}) > 0 and dt_static(P_{x86}, P'_{x86}) > 0 $\implies$ both programs are still different when compiled to machine code.*

If the property fits for two programs, then the underlying compiler does not remove the transformations made by CROW. Notice that, this property only makes sense between variants of the same program, including the original.

TODO Improve this !

Metric 4 *Preservation ratio*

Given a program P and a corpus of variants V generated by CROW from P.

$$preservation\_ratio = \frac{|v_1, v_2 \in V \cap \{P\}, \forall v_1, v_2 \ ensuring \ Property \ 1|}{|V \cap \{P\}|^2}$$

Notice that Metric 4 implies a pairwise comparison between all variants and the original program.

We only take into account the x86 representation after the WebAssembly code is compiled to the machine code. This decision is not arbitrary, according to the study of **TODO** Paper on binary diff survey , any conclusion carried out by comparing two program binaries under a specific target can be extrapolated to another target for the same binaries.

2.2 Evaluation

To answer RQ2 we use the same corpora proposed and evaluated in chapter 1, **CROW prime** and **MEWE prime**. We analyze the variants generated in the RQ1 answering. **TODO** Add the numbers here

2.2.1 Static comparison

For each function on the corpora, we compare the sequence of instructions of each variant with the initial program and the other variants. We obtain the **??** values for each program-variant WebAssembly pair code. We compute the DTW distances with STRAC [**?**].

2.2.2 Dynamic comparison

To compare program and variants behavior during runtime, we analyze all the unique program variants generated by CROW in a pairwise comparison. We use SWAM[1] to collect the function and instruction traces. SWAM is a WebAssembly interpreter that provides functionalities to capture the dynamic information of WebAssembly program executions including the stack operations. We compute the DTW distances with STRAC [**?**].

Furthermore, we collect the execution time, Metric 3, for all programs and their variants. We execute each program or variant **TODO** XXX times and we compare the collected execution times using a Mann-Withney test [?] .

2.2.3 Preservation

We collect Metric 4 for all programs and their generated variants. We use the engines listed in Table 2.1.

TODO We can add the other binaries

Name	Properties
V8 [?]	V8 is the engine used by Chrome and NodeJS to execute JavaScript and WebAssembly. **TODO** Explain compilation process
wasmtime [?]	Wasmtime is a standalone runtime for WebAssembly. This engine is used by the Fastly platform to provide Edge-Cloud computing services. **TODO** Explain compilation process

Table 2.1: Wasm engines used during the diversification assessment study. The table is composed by the name of the engine and the description of the compilation process for them.

[1] `https://github.com/satabin/swam`

2.3 Results

2.3.1 Static

2.3.2 Dynamic

2.3.3 Preservation

We translate each WebAssembly multivariant binary with Lucet, to determine the impact of this translation to machine code on the function variants and the diversity of paths in the multivariant call graph.

The 'x86 code' section of **??** summarizes the key data to answer RQ2. Column #Variants shows the number of preserved variants in the x86 code of each endpoint, column #Paths shows the number of possible paths in the x86 multivariant binary. The last two columns show the ratio of paths (PP) and variants (PV) preserved in x86. Note that the path preservation ratio metric is a projection of the variant preservation and the call graph in the multivariant binary.

In all cases, more than 77% of the individual function variants present in the multivariant Wasm binary are preserved in the x86 multivariant. This high preservation rate for function variants allows to preserve a large ratio of possible paths in the multivariant call graph. In 4 out of 7 cases, more than 83% of the possible execution paths in the multivariant Wasm binary are preserved. The translation to machine code preserves 21% and 17% of the possible paths for `qr_str` and `qr_image`. Yet, the x86 version of the multiversion call graph for these endpoints still includes millions of possible paths, with 17 and 15 randomization points. The translation to machine drastically reduces the potential for randomized execution paths only for `bin2base64`, for which it preserves only 25% of the possible paths, for a total of 41 paths.

We have identified why some variants are not preserved the translation from Wasm to x86. Lucet performs optimization passes before generating machine code. In some cases, this can annihilate the effect of CROW's diversification transformation. For example, in Listing 2.1, CROW synthesizes a variant in the right column by splitting it in two multiplications relying on the integer overflow mechanism. A constant merging optimization pass could remove the constant multiplications by performing it at compilation time. The other transformation cases that we have observed have the same property, the transformations are simple enough to be quickly verified at compilation time.

We identified where the optimizations are done in Lucet's compiler, [2]. It performs optimization-like transformations that are simpler than the ones introduced by CROW. With this result we also encourage to avoid the usage of the insertion of `nop` instructions either in Wasm or machine code. `nop` operations could be easily detected and removed by a latter optimization stage.

[2] `https://github.com/bytecodealliance/wasmtime/blob/main/cranelift/codegen/src/preopt.peepmatic` and `https://github.com/bytecodealliance/wasmtime/blob/main/cranelift/codegen/src/postopt.rs`

```
; previous stack code;
i32.const -10
i32.mul
```

```
; previous stack code ;
i32.const -1931544174
i32.mul
i32.const 109653155
i32.mul
```

Listing 2.1: Two examples of block variants that are functionally equivalent and implement with different WebAssembly instructions. The variant on the left, generated by CROW, is not preserved through the translation to machine code.

Moreover, the last three endpoints have a path preservation ratio that is less than 0.25, even with more than 87% of individual function variants that are preserved. This is explained by the fact that the number of possible paths is related to both the number of variants and to the complexity of the call graph.

TODO Add image

The example in **??** illustrates this phenomenon. Suppose an original binary composed of three functions with the call graph illustrated at the top of the figure. Here, we count 2 possible paths (one with no iteration, and one with a single iteration). CROW generates 2 variants for $f2$ and 4 variants for $f3$, the multivariant wasm call graph is illustrated at the center of the figure. The number of possible execution paths increases to 40. In the translation process, Lucet transforms the two WebAssembly function variants for $f2$ into the same x86 function. In this case, the number of possible execution paths in the x86 multivariant call graph is reduced by a factor of 2, from 40 to 20. However, the number of variants is decreased only in 1. The complexity of the call graph has a major impact on the number of possible execution paths.

The translation from WebAssembly to machine code through Lucet preserves a high ratio of function variants. This leads to the preservation of high numbers of possible execution paths in the multivariant binaries. Our multivariant execution scheme is appropriate for the state-of-the-art runtime of edge computing nodes.

2.4 Conclusions